# System Security & x86 Assembly Fundamentals

## Course Introduction

# Alexis Ahmed

Senior Penetration Tester @HackerSploit

Offensive Security Instructor @INE

# Course Topic Overview

- Architecture Fundamentals
  - CPU Architecture & Components
  - CPU Registers
  - General Purpose Registers
  - Process Memory
  - The Stack
- X86 Assembly Fundamentals
  - Assemblers & Compilers
  - Introduction to x86 Assembly
  - Basic x86 Assembly Programming

# Prerequisites

- Basic understanding of computer architecture concepts.
- Familiarity with low-level programming languages such as C or C++.
- Proficiency in using a Linux-based operating system, including basic command-line navigation and file manipulation.
- Some knowledge of assembly language programming concepts would be beneficial but not required.

# Learning Objectives:

1. Understanding Computer Architecture Fundamentals:
    - Explain the basic principles of computer architecture, including CPU architecture and process memory.
    - Understand the role of the CPU, CPU registers and memory in the operation of a computer system.
2. IA-32 CPU Architecture:
    - Describe the architecture of IA-32 processors, including registers, instruction set, and execution pipeline.
    - Understand the function and purpose of general-purpose registers, and other CPU components.
3. Memory Organization and Process Memory:
    - Explore the different memory regions, including code, data, heap, and stack, and their roles in program execution.
4. Understanding the Stack and Stack Frames:
    - Explain the stack data structure and its role in function calls and local variable storage.
    - Understand the concept of stack frames and their organization in memory during function execution.
5. Introduction to IA-32 Assembly Language:
    - Define assembly language and its role in low-level programming.
    - Differentiate between high-level languages and assembly language.
6. IA-32 Assembly Language Basics:
    - Learn how to use assembly language instructions for basic arithmetic and logical operations.
    - Understand data movement instructions and memory addressing modes in IA-32 assembly.

# Let’s Get Started!

# Introduction To System Security

# Introduction

- Welcome to the System Security section of the PTP learning path!
- System Security is a comprehensive course designed to equip penetration testers and red teamers with the foundational knowledge of computer architecture, assembly language, and security mechanisms.
- The purpose of this course is to give you the fundamental knowledge needed to help you improve your skills in topics such as fuzzing, exploit development, buffer overflows, debugging, reverse engineering and malware analysis. (These topics will be explored in-detail in the Exploit Development Course)

# Introduction

- This course will provide you with an understanding of CPU architecture, memory management, and the intricacies of x86 and x64 instruction sets.

- You will also learn the basics of 32-bit assembly language for the Intel Architecture (IA-32) family of processors on Linux and learn how to apply it to Infosec.

- Once we are done with the basics, we will use what we have learned in the Exploit Development course where we will be exploring Buffer Overflows, fuzzing, shellcoding etc.

# Introduction

- Assembly language is the gateway to understanding exploitation techniques, reverse engineering, shellcoding and other low level fields in information security.

- Without a good grasp of system architecture and assembly knowledge it will not be possible to master these fields effectively.

# CPU Architecture

# CPU Architecture

- A CPU (Central Processing Unit) is often referred to as the brain of a computer, responsible for executing instructions and performing calculations.
- The Central Processing Unit (CPU) is the device in charge of executing the machine code of a program.
- The machine code, or machine language, is the set of instructions that the CPU processes.

# CPU Architecture

- Each instruction is a primitive command that performs a specific operation such as moving data between registers, working with memory, changing the execution flow of a program, and performing arithmetic and more.

- As a rule, each CPU has its own instruction set architecture (ISA).

- CPU instructions are represented in hexadecimal (HEX) format. Due the inherent unreadable nature and complexity, it is impossible for humans to read or utilize it in its natural format.

# CPU Architecture

- As a result, machine code is translated into Assembly Language (ASM). Assembly language (ASM) is mnemonic code (a more readable language) that humans can understand and interpret.

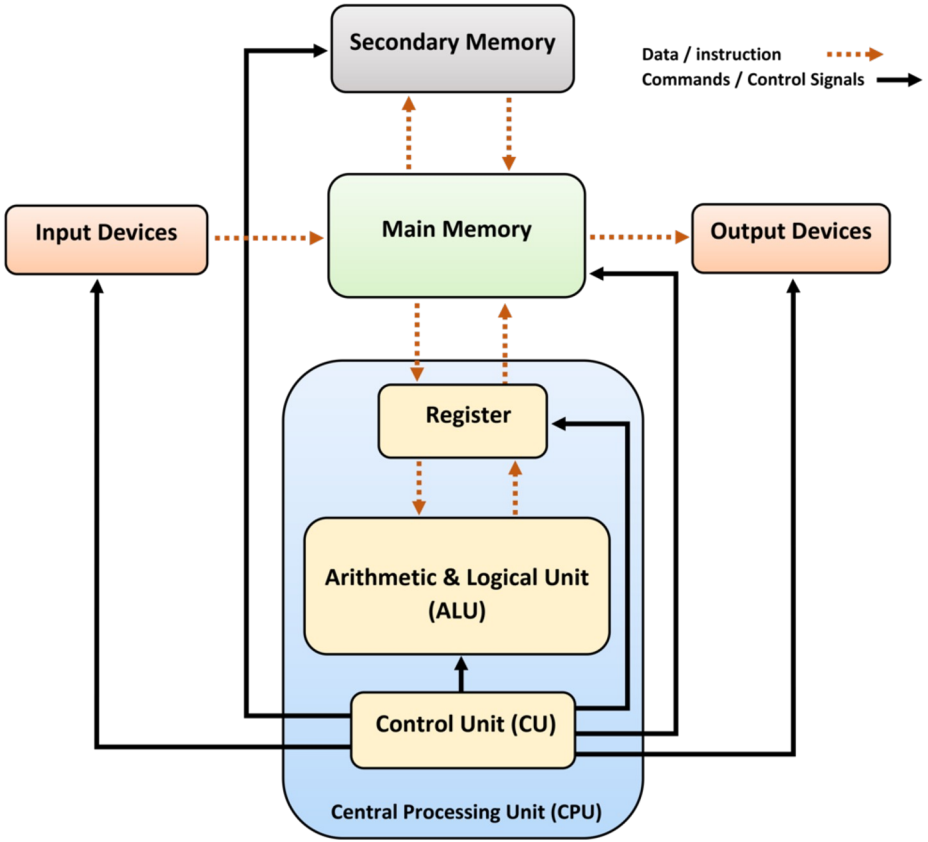
- Assembly language is a low-level programming language that is closely related to the machine code instructions of a specific CPU architecture.

- It provides a symbolic representation of the machine instructions and allows programmers to write instructions using mnemonics and symbolic labels rather than binary code.

# CPU Architecture

- Assembly language is closely tied to the architecture of the target CPU.
- Different CPU architectures have their own instruction sets and assembly languages.
- For example, x86 assembly language is used for Intel and AMD processors, while ARM assembly language is used for ARM-based processors.

# CPU Components



# CPU Components

## Control Unit (CU)

- The Control Unit is responsible for coordinating and controlling the operations of the CPU.
- It fetches instructions from memory, decodes them, and manages the execution of instructions by directing data movement and control flow within the CPU.

## Arithmetic Logic Unit (ALU)

- The Arithmetic Logic Unit is the component responsible for performing arithmetic and logical operations.
- It can perform basic operations like addition, subtraction, multiplication, division, as well as logical operations like AND, OR, and NOT.

# CPU Components

## Registers

- Registers are small, high-speed storage locations within the CPU used to store data temporarily during processing. Some common types of registers include:
  - Program Counter (PC): Holds the memory address of the next instruction to be fetched.
  - Instruction Register (IR): Holds the currently executing instruction.
  - Accumulator: Stores the result of arithmetic and logic operations.
  - General-Purpose Registers: Used to store intermediate values and operands during instruction execution.

# CPU Instruction Set Architecture (ISA)

- Each CPU has its own instruction set architecture (ISA). The ISA is the set of instructions that a programmer (or a compiler) must understand and use to write a program correctly for that specific CPU and machine.

- In other words, ISA is what a programmer can see: memory, registers, instructions, etc. It provides all the necessary information for who wants to write a program in that machine language.

# CPU Instruction Set Architecture (ISA)

- The most common ISA is the x86 instruction set (or architecture) which originated from the Intel 8086 processor.

- The x86 acronym identifies 32-bit processors, while x64 (aka x86_64 or AMD64) identifies the 64-bit versions.

# Registers

# Registers

- CPU registers are small, high-speed storage locations located within the CPU (Central Processing Unit).
- They are used to temporarily hold data that is being processed or manipulated by the CPU.
- Registers play a crucial role in the execution of instructions and the management of data within the computer system.

# Registers

- The architecture of the CPU (32-bit and 64-bit), refers to the width/size of the CPU registers.
- Each CPU has its fixed set of registers that are accessed when required.
- You can think of registers as temporary variables used by the CPU to retrieve and store data.
- For the purpose of this course, we will focus on a specific group of registers: General Purpose Registers (GPRs).

# General Purpose Registers

- General-purpose registers are used to store data temporarily during program execution.
- They are versatile and can hold various types of data, such as integers, memory addresses, or intermediate results of arithmetic/logical operations.
- Examples of general-purpose registers include:
  - EAX, EBX, ECX, EDX: Used for general data manipulation and arithmetic operations.
  - ESI, EDI: Often used for string manipulation operations.
  - ESP, EBP: Used for managing the stack (ESP for stack pointer, EBP for base pointer).
  - In 64-bit architectures (x64), the general-purpose registers are extended to 64 bits (RAX, RBX, RCX, RDX, etc.), providing increased addressable memory space and support for larger data types.

# Registers

- The table in the next slide summarizes the eight general purpose registers.
- Pay close attention to the x86 naming convention as it refers to the register names for the x86 (32-bit) architecture.
- We will see how the names differ for 64-bit, 32-bit, 16-bit and 8-bit as we progress in the slides.

# X86 Registers

| X86 NAME | NAME | PURPOSE |
|---|---|---|
| **EAX** | **Accumulator** | Used in arithmetic operations |
| **ECX** | **Counter** | Used in shift/rotate instruction and loops |
| **EDX** | **Data** | Used in arithmetic operation and I/O |
| **EBX** | **Base** | Used as a pointer to data |
| **ESP** | **Stack Pointer** | Pointer to the top of the stack |
| **EBP** | **Base Pointer** | Pointer to the base of the stack (aka Stack Base Pointer, or Frame pointer) |
| **ESI** | **Source Index** | Used as a pointer to a source in stream operation |
| **EDI** | **Destination** | Used as a pointer to a destination in stream operation |

# General Purpose Registers

**EAX (Accumulator Register)**

- The EAX register is the primary accumulator register used for arithmetic and logic operations. It is often used to store operands and receive results of computations. In certain contexts, it holds function return values and is used as a scratch register for intermediate calculations.

**EBX (Base Register)**

- The EBX register, also known as the base register, is typically used as a pointer to data in memory or as a base address for memory operations. It can also serve as a general-purpose register for storing data temporarily during computations.

# General Purpose Registers

**ECX (Counter Register)**

- The ECX register, known as the counter register, is commonly used for loop control and iteration counting. It is often used in conjunction with the LOOP instruction to implement loops and repetitive tasks.

**EDX (Data Register)**

- The EDX register, or data register, is used in conjunction with EAX for certain arithmetic operations that require a wider range of data storage (e.g., 64-bit multiplication and division). It can also serve as a general-purpose register for storing data.

# General Purpose Registers

**ESI (Source Index Register)**

- The ESI register, known as the source index register, is commonly used in string manipulation operations. It typically holds the starting address of the source data or the source string during operations like copying, comparing, or searching strings.

**EDI (Destination Index Register)**

- The EDI register, or destination index register, complements the ESI register in string manipulation operations. It usually holds the starting address of the destination data or the destination string during string operations like copying or concatenation.

# General Purpose Registers

**ESP (Stack Pointer Register)**

- The ESP register, or stack pointer register, points to the top of the stack in memory. It is used to manage the stack, a special area of memory used for storing function parameters, local variables, return addresses, and other data during program execution.

**EBP (Base Pointer Register)**

- The EBP register, known as the base pointer register, is commonly used in conjunction with the ESP register to access parameters and local variables within function calls. It serves as a reference point for accessing data stored on the stack.

# Architecture Specific Register Names

- Although this information may seem overwhelming, everything will become make sense once we explore the Stack.
- The naming convention of the old 8-bit CPU had 16-bit registers divided into two parts:
  - A low byte, identified by an L at the end of the name, and
  - A high byte, identified by an H at the end of the name.
- The 16-bit naming convention combines the L and the H, and replaces it with an X. While for the Stack Pointer, Base Pointer, Source and Destination registers it simply removes the L.
- In the 32-bit representation, the register acronym/name is prefixed with an E, meaning extended. Whereas, in the 64-bit representation, the E is replaced with the R.

# Architecture Specific Register Names

- The table in the next slide outlines the naming conventions for 8-bit, 16-bit, 32-bit and 64-bit registers.

- Although we will mainly be using the 32-bit naming convention throughout the duration of this course, it is useful to understand the 64-bit name convention as well.

# Architecture Specific Register Names

| Register | Accumulator | Counter | Data | Base |
|---|---|---|---|---|
| 64-bit | RAX | RCX | RDX | RBX |
| 32-bit | EAX | ECX | EDX | EBX |
| 16-bit | AX | CX | DX | BX |
| 8-bit | AH AL | CH CL | DH DL | BH BL |

| Register | Stack Pointer | Base Pointer | Source | Destination |
|---|---|---|---|---|
| 64-bit | RSP | RBP | RSI | RDI |
| 32-bit | ESP | EBP | ESI | EDI |
| 16-bit | SP | BP | SI | DI |
| 8-bit | SPL | BPL | SIL | DIL |

# Instruction Pointer (EIP)

- In addition to the eight general purposes registers, there is also another register that will be important for what we will be doing in this course, the EIP (x86 naming convention).

- The Instruction Pointer (EIP) controls the program execution by storing a pointer to the address of the next instruction (machine code) that will be executed.

**NOTE**

***It tells the CPU where the next instruction is.***

# Process Memory

# Process Memory

- Process memory management is a fundamental aspect of operating systems, responsible for organizing and managing memory resources for running programs (processes).

- Modern operating systems implement a concept known as virtual memory, which abstracts physical memory resources and presents each process with a virtual address space.

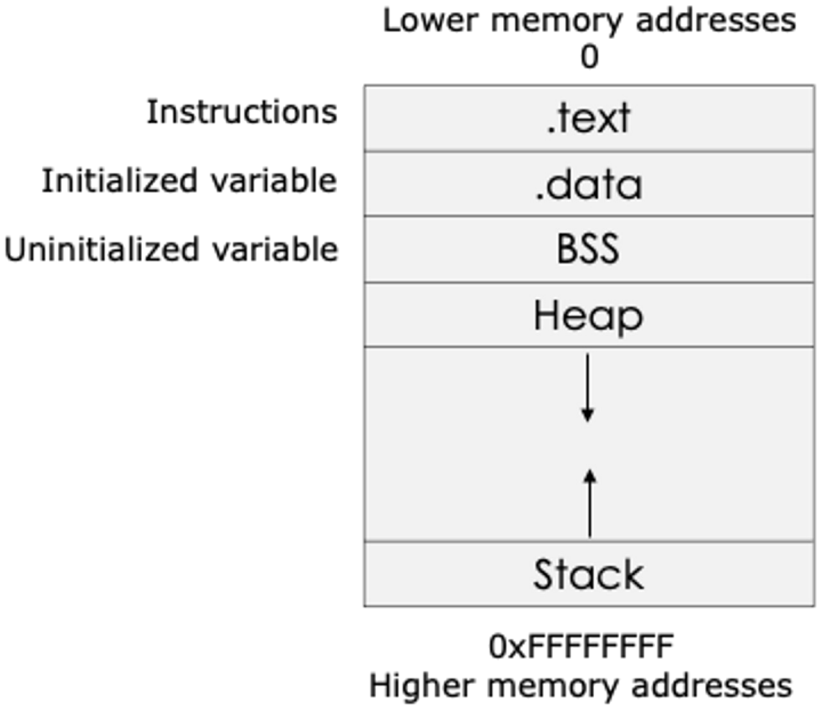
- Each process perceives its memory as a contiguous block of memory addresses, starting from address 0 and extending up to the maximum addressable space.

# Memory Segmentation

- Process memory is typically divided into segments, each serving a specific purpose:
  - Code Segment: Contains the executable code of the program.
  - Data Segment: Stores initialized data, such as global variables and static variables.
  - BSS Segment: Contains uninitialized data, initialized to zero during program execution.
  - Heap Segment: Dynamically allocated memory for program data structures (e.g., heap memory allocated using functions like malloc() and free()).
  - Stack Segment: Stores function call frames, local variables, and function parameters. The stack grows and shrinks dynamically as functions are called and return.

# Memory Segmentation

When a process runs, it is typically organized in memory as shown in the figure on the right.



# Memory Segmentation

- The Text region, or instruction segment, is fixed by the program and contains the program code (instructions).
- This region is marked as read-only since the program should not change during execution.
- The Data region is divided into initialized data and uninitialized data. Initialized data includes items such as static and global declared variables that are predefined and can be modified.
- The uninitialized data, named Block Started by Symbol (BSS), also initializes variables that are initialized to zero or do not have explicit initialization (ex. static int t).

# Memory Segmentation

- Next is the Heap, which starts right after the BSS segment. During the execution, the program can request more space in memory via *brk* and *sbrk* system calls, used by *mlloc*, *realloc* and *free*.

- Hence, the size of the data region can be extended; this is not vital, but if you are very interested in a more detailed process, these may be topics to do your own research on.

- The last region of the memory is the Stack. For our purposes, this is the most important structure we will deal with.

# Understanding The Stack

# The Stack

- The stack is a fundamental data structure in computer science and plays a crucial role in the execution of programs.

- In the context of process memory management, the stack is a region of memory allocated for storing function call frames, local variables, function parameters, and return addresses during program execution.

# LIFO (Last-In-First-Out)

- The stack follows the Last-In-First-Out (LIFO) principle, meaning that the last item pushed onto the stack is the first one to be popped off.
- This makes the stack an ideal data structure for managing function calls and storing temporary data that needs to be accessed in reverse order of its insertion.
- You can think of the stack as an array used for saving a function's return addresses, passing function arguments, and storing local variables.
- The purpose of the ESP register (Stack Pointer) is to identify the top of the stack, and it is modified each time a value is pushed in (PUSH) or popped out (POP).

# The Stack Pointer

- The stack is managed by a special CPU register called the stack pointer (ESP in x86 architectures, RSP in x64 architectures).
- The stack pointer points to the top of the stack, indicating the location in memory where the next item will be pushed or popped.
- As items are pushed onto the stack (e.g., during function calls), the stack pointer is decremented to allocate more space. Conversely, when items are popped off the stack (e.g., when functions return), the stack pointer is incremented to reclaim memory.

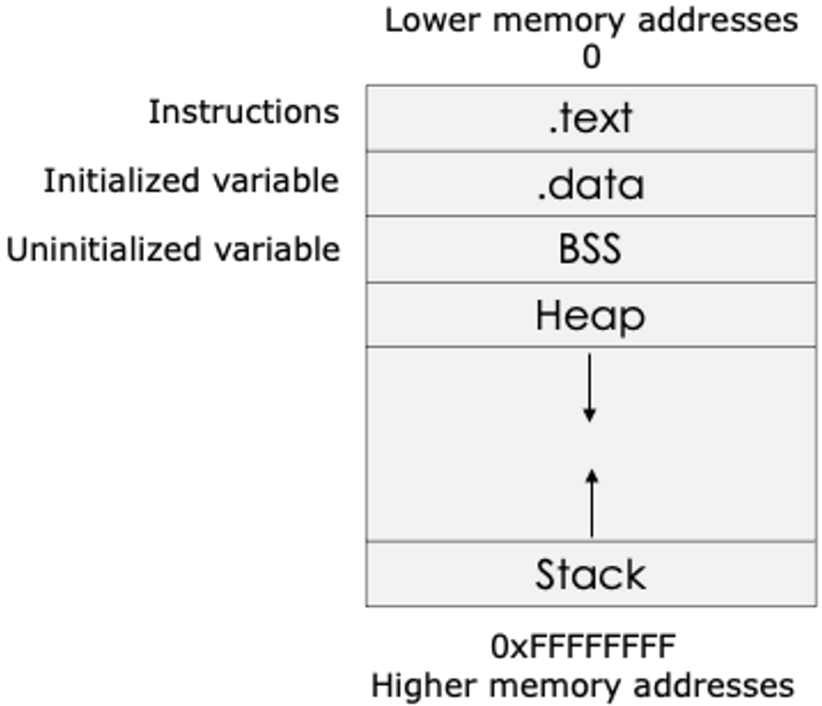
# The Stack Pointer

**ESP (Stack Pointer Register)**

- The ESP register, or stack pointer register, points to the top of the stack in memory.
- It is used to manage the stack, a special area of memory used for storing function parameters, local variables, return addresses, and other data during program execution.

# The Stack

When a process runs, it is typically organized in memory as shown in the figure on the right.



# The Stack

- Before exploring how the stack works and how to operate on it, it is important to understand how the stack grows.
- Common sense would make you think that the stack grows upwards, towards higher memory addresses, but as you saw in the previous memory structure diagram, the stack grows downward, towards the lower memory addresses.
- This is probably due to historical reasons when the memory in old computers was limited and divided into two parts: Heap and Stack.
- Knowing the limits of the memory allowed the programmer to know how big the heap and/or the stack would be.

# The Stack

- It was decided that the Heap would start from lower addresses and grow upwards and the Stack would start from the end of the memory and grow downward.

# The Stack

- As previously mentioned, the stack is a LIFO structure, and the most fundamental operations are the **PUSH** and **POP**.

- The main pointer for these operations is the ESP, which contains the memory address for the top of the stack and changes during each PUSH and POP operation.

- "Push" and "pop" are two fundamental operations performed on a stack data structure. Here's an explanation of each:

# PUSH

- "Push" is an operation that adds an element to the top of the stack. When you push an item onto the stack, it becomes the new top element, and the stack grows in size.
- The push operation involves two main steps:
    - Increment the stack pointer: The stack pointer is moved upwards to allocate space for the new element.
    - Store the element: The new element is stored at the memory location indicated by the stack pointer.
- In programming terms, pushing an element onto the stack typically involves copying the value of the element into the memory location pointed to by the stack pointer and then updating the stack pointer to point to the next available memory location.

# POP

- "Pop" is an operation that removes the top element from the stack. When you pop an item from the stack, it is removed from the top, and the stack shrinks in size.

- The pop operation also involves two main steps:
  - Access the top element: The element at the top of the stack is retrieved.
  - Decrement the stack pointer: The stack pointer is moved downwards to deallocate the space previously occupied by the top element.

- In programming terms, popping an element from the stack typically involves reading the value stored at the memory location pointed to by the stack pointer, then decrementing the stack pointer to point to the next element on the stack.

# PUSH Example 1

# PUSH Example 1

- The first example of how the stack changes is the execution of the following instruction: **PUSH E**.
- The second example is the execution of the following instruction: **POP E**.

# PUSH Example 1

**PUSH Instruction:**
PUSH E

**PUSH Process:**
A PUSH is executed, and the ESP register is modified.

**Starting value:**
The ESP points to the top of the stack.

# PUSH Example 1

## PROCESS

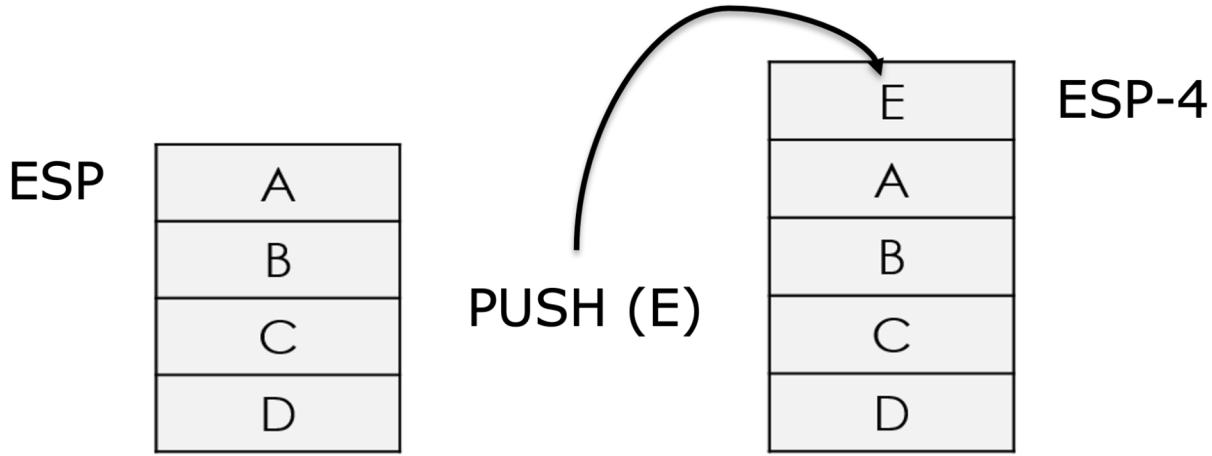
- A PUSH instruction subtracts 4 (in 32-bit) or 8 (in 64-bit) from the ESP and writes the data to the memory address in the ESP, and then updates the ESP to the top of the stack.

- Remember that the Stack grows backward. Therefore the PUSH subtracts 4 or 8, in order to point to a lower memory location on the stack. If we do not subtract it, the PUSH operation will overwrite the current location pointed by ESP (the top) and we would lose data.

# PUSH Example 1

## ENDING VALUE

- The ESP points to the top of the stack -4.



# PUSH Example 2

# PUSH Example 2

Now for a more detailed example of the PUSH instruction.

**Starting value: (ESP contains the address value)**

- ESP points to the following memory address: `0x0028FF80`.

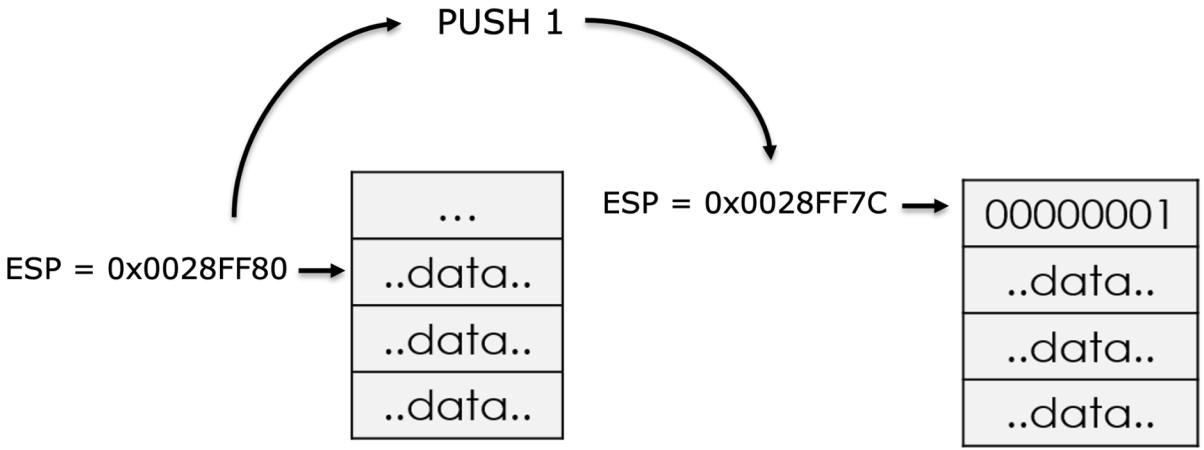
**Process:**

- The program executes the instruction PUSH 1.
- ESP decreases by 4, becoming `0x0028FF7C`, and the value 1 will be pushed on the stack.

# PUSH Example 2

## ENDING VALUE

- ESP points to the following memory address: 0x0028FF7C.



# POP Example

# POP Example 1

**Starting value: (ESP contains the address value)**

- After the PUSH 1, the ESP points to the following memory address: 0x0028FF7C.

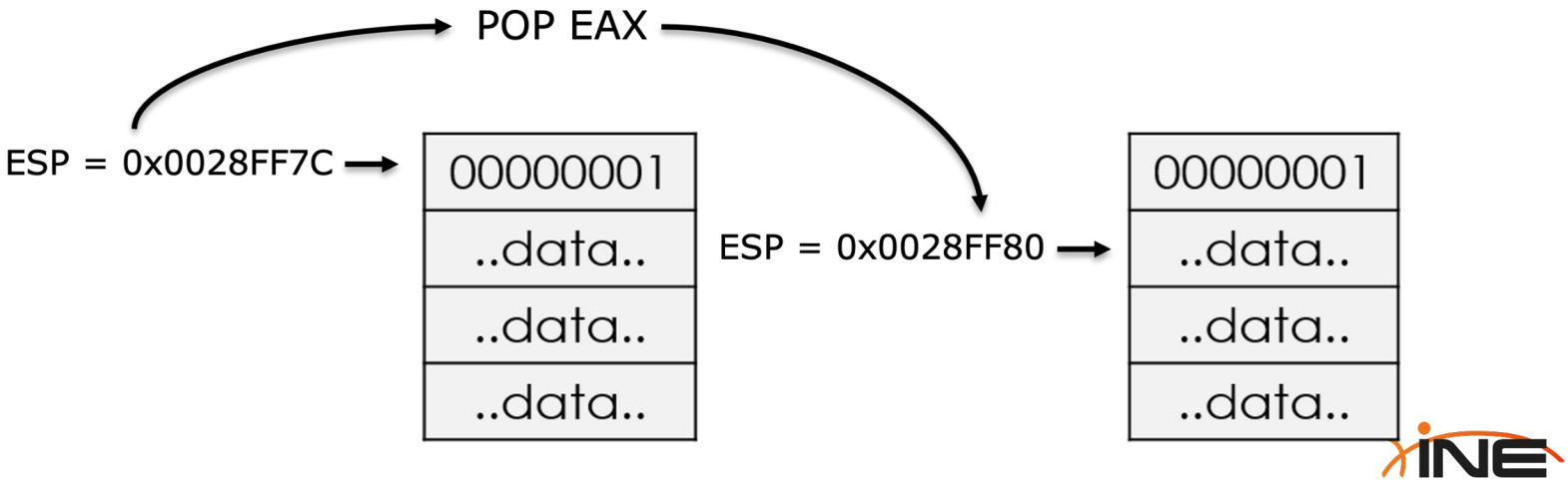
**Process:**

- The program executes the inverse instruction, POP EAX. The value (00000001) contained at the address of the ESP (0x0028FF7c = the top of the Stack), will be popped out from the stack and will be copied in the EAX register.
- Then, ESP is updated by adding 4 and becoming 0x0028FF80.

# POP Example 1

## ENDING VALUE

- ESP points to the following memory address: 0x0028FF8. It returns to its original value.



# Stack Frames

# Stack Frames

- A stack frame, also known as an activation record or call frame, is a data structure used by the CPU and the operating system to manage function calls and execution flow within a program.

- It contains information related to a single function call, including parameters, local variables, return address, and other relevant data.

# Procedures & Functions

- Now that we know more about the Stack, we will investigate how procedures and functions work. It is important to know that procedures and functions alter the normal flow of the process.

- When a procedure or a function terminates, it returns control to the statement or instruction that called the function.

# Procedures & Functions

- Functions contain two important components, the prologue and the epilogue, which we will discuss later, but here is a very quick overview.
- The prologue prepares the stack to be used, similar to putting a bookmark in a book. When the function has completed, the epilogue resets the stack to the prologue settings.
- The Stack consists of logical stack frames (portions/areas of the Stack), that are **PUSHed** when calling a function and **POPped** when returning a value.

# Procedures & Functions

- When a subroutine, such as a function or procedure, is started, a stack frame is created and assigned to the current ESP location (top of the stack); this allows the subroutine to operate independently in its own location in the stack.

- When the subroutine ends, two things happen:
  - The program receives the parameters passed from the subroutine.
  - The Instruction Pointer (EIP) is reset to the location at the time of the initial call.

# Procedures & Functions

- In other words, the stack frame keeps track of the location where each subroutine should return the control when it terminates.
- We will break down this process in a more specific example for you to better understand how stack frames work.
- First, we will explain the operations, and then we will illustrate how it happens in an actual program.  When this program is run, the following process occurs.

# Example

# Example

This process has three main operations:

1. When a function is called, the arguments [(in brackets)] need to be evaluated.

1. The control flow jumps to the body of the function, and the program executes its code.

1. Once the function ends, a return statement is encountered, the program returns to the function call (the next statement in the code).

# Example

The following code snippet explains how this works in the Stack. This example is written in C:

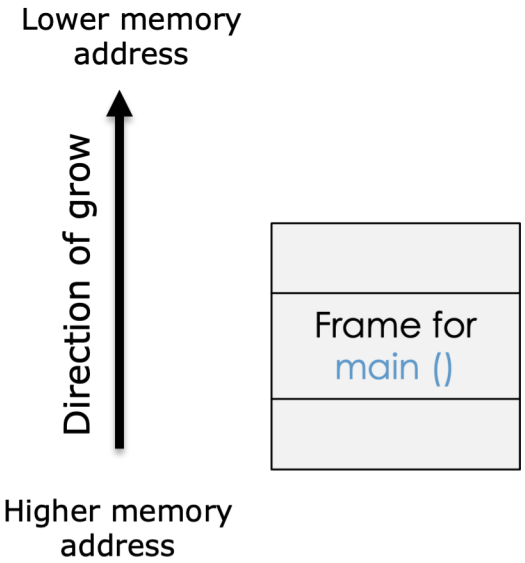
```
int b(){ //function b
        return 0;
}
int a(){ // function a
        b();
        return 0;
}
int main (){//main function
        a();
        return 0;
}
```

# Step 1

- The entry point of the program is main().
- The first stack frame that needs to be pushed to the Stack is the main() stack frame.
- Once initialized, the stack pointer is set to the top of the stack and a new main() stack frame is created.

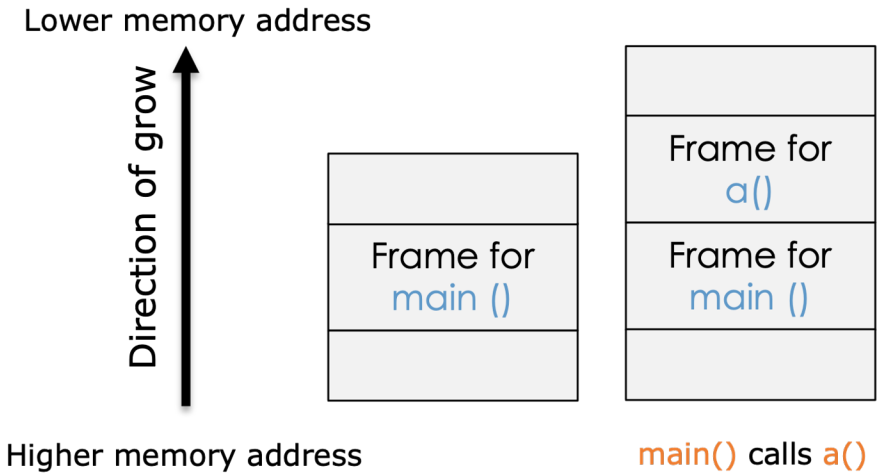
# Step 1

- Our stack will then look like the following:



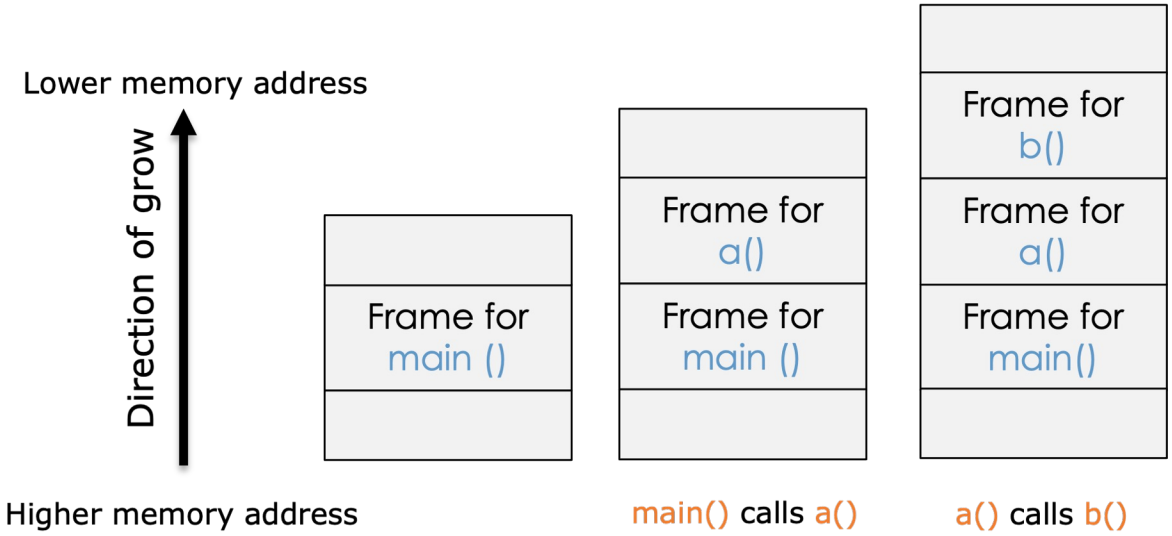
# Step 2

- Once inside main(), the first instruction that executes is a call to the function named a().
- Once again, the stack pointer is set to the top of the stack of main() and a new stack frame for a() is created on the stack.



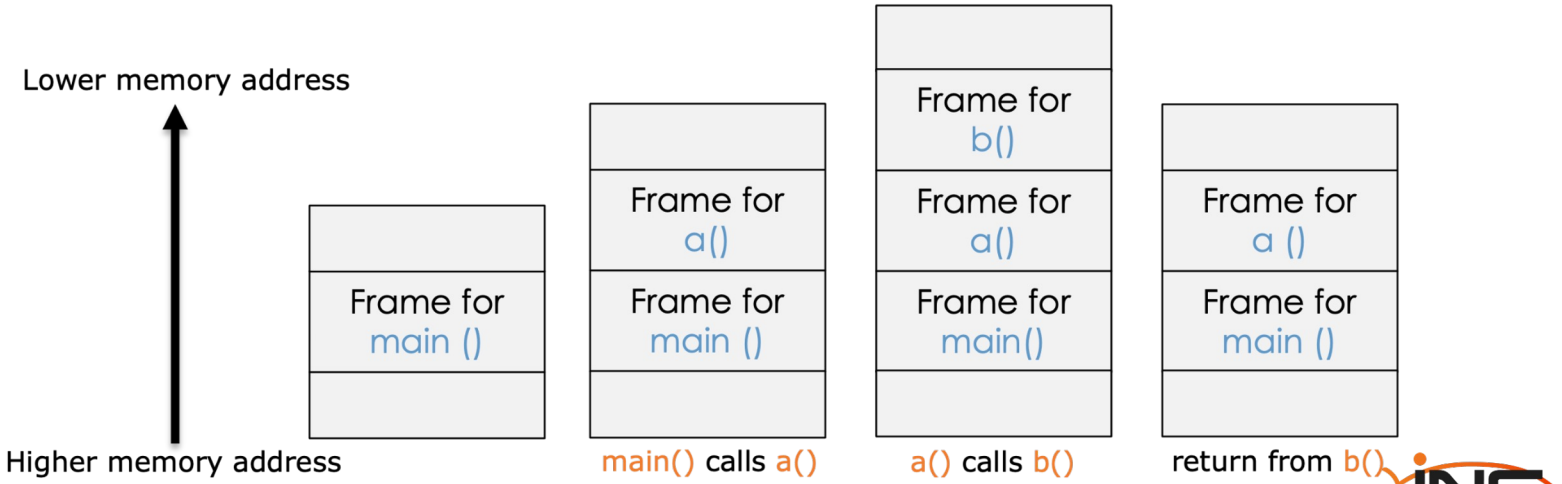
# Step 3

- Once the function a() starts, the first instruction is a call to the function named b(). Here again, the stack pointer is set, and a new stack frame for b() will be pushed on the top of the stack.



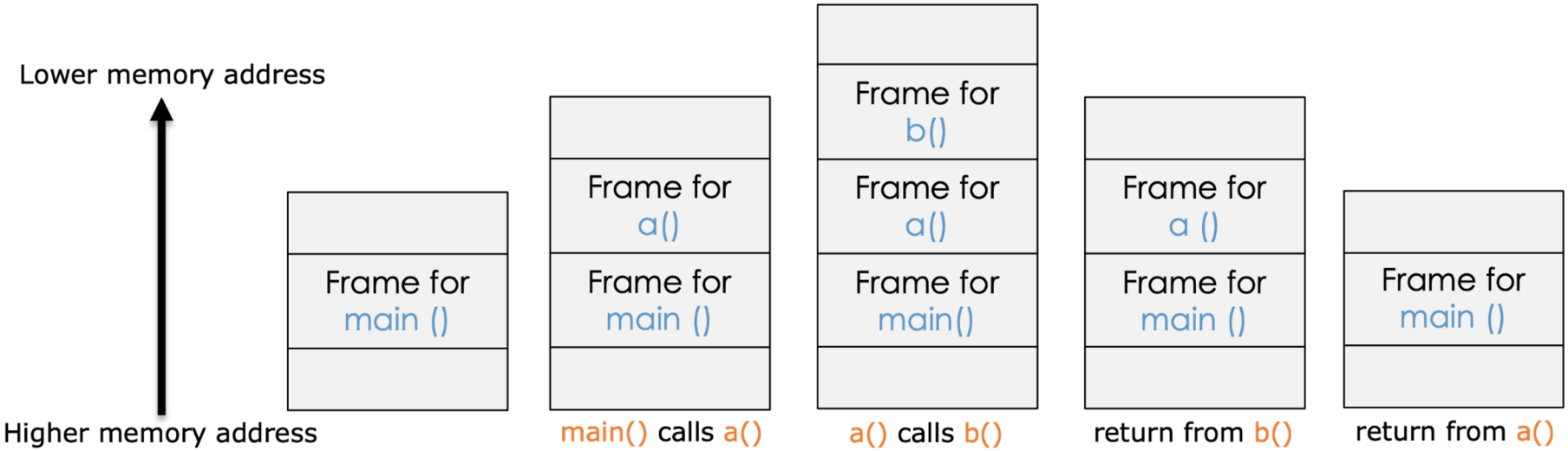
# Step 4

- The function b() does nothing and just returns. When the function completes, the stack pointer is moved to its previous location, and the program returns to the stack frame of a() and continues with the next instruction.



# Step 5

- The next instruction executed is the return statement contained in a(). The a() stack frame is popped, the stack pointer is reset, and we will get back in the main() stack frame.



# Conclusion

- This was a quick overview of how stack frames work, but for buffer overflows, we need to go into more detail as to what information is stored, where it is stored and how the registers are updated.

# Assemblers & Compilers

# Assemblers

- In the context of assembly language programming, an assembler is a type of language translator that converts assembly language code into machine code, which is directly executable by the computer's CPU.

- Assembly language is a low-level programming language that uses mnemonic instructions to represent machine instructions, making it easier for humans to read and write compared to raw binary machine code.

# Assemblers

- There are several types of assemblers that depend on the target system's ISA (Instruction Set Architecture):
  - Microsoft Macro Assembler (MASM), x86 assembler that uses the Intel syntax for MS-DOS and Microsoft Windows
  - GNU Assembler (GAS), used by the GNU Project, default back-end of GCC.
  - Netwide Assembler (NASM), x86 architecture used to write 16-bit, 32-bit (IA-32) and 64-bit (x86-64) programs, one of the most popular assemblers for Linux
  - Flat Assembler (FASM), x86, supports Intel-style assembly language on the IA-32 and x86-64

# How Assemblers Work

- When source code is assembled, the resulting file is called an object file. It is a binary representation of the program.

- While the assembly instructions and the machine code have a one-to-one correspondence, and the translation process may be simple, the assembler does some further operations such as assigning memory location to variables and instructions and resolving symbolic names.

- Once the assembler has created the object file, a linker is needed in order to create the actual executable file. What a linker does is take one or more object files and combine them to create the executable file.

# How Assemblers Work

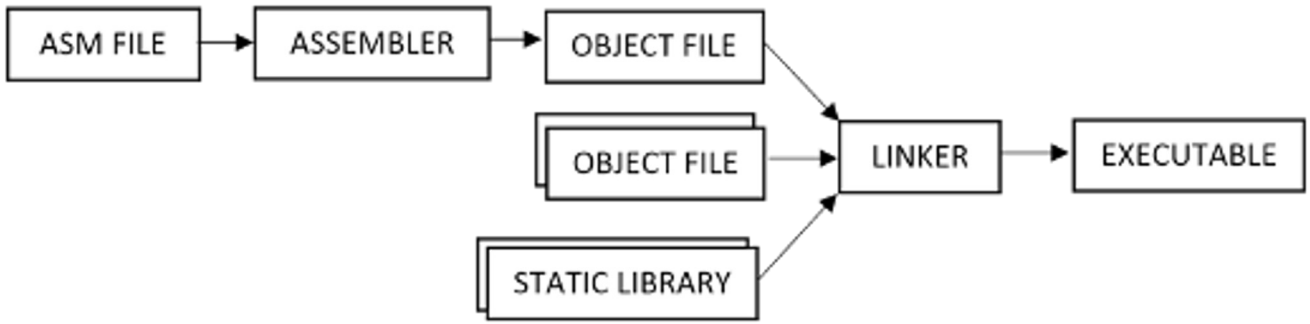
- When source code is assembled, the resulting file is called an object file. It is a binary representation of the program.

- While the assembly instructions and the machine code have a one-to-one correspondence, and the translation process may be simple, the assembler does some further operations such as assigning memory location to variables and instructions and resolving symbolic names.

- Once the assembler has created the object file, a linker is needed in order to create the actual executable file. What a linker does is take one or more object files and combine them to create the executable file.

# How Assemblers Work

- The process from the Assembly code to the executable file can be illustrated as follows:



# Compilers

- A compiler is a software tool used in computer programming to translate source code written in a high-level programming language into machine code or executable code that can be directly executed by a computer's CPU.

- The compiler is similar to the assembler. It converts high-level source code (such as C) into low-level code or directly into an object file. Therefore, once the output file is created, the previous process will be executed on the file. The end result is an executable file.

# Introduction To Assembly

# Assembly Language

- Assembly language is a low-level programming language that is closely related to the machine code instructions of a specific CPU architecture.
- It provides a symbolic representation of the machine instructions and allows programmers to write instructions using mnemonics and symbolic labels rather than binary code.
- Different CPU architectures have their own instruction sets and assembly languages.
- For example, x86 assembly language is used for Intel and AMD processors, while ARM assembly language is used for ARM-based processors.

# Assembly Language

## Assembly Language

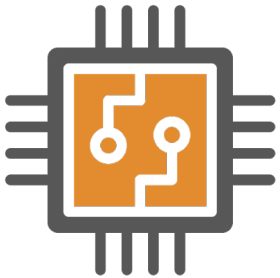
```
mov eax, ebx
add eax, ebx
```

Assembler + Linker

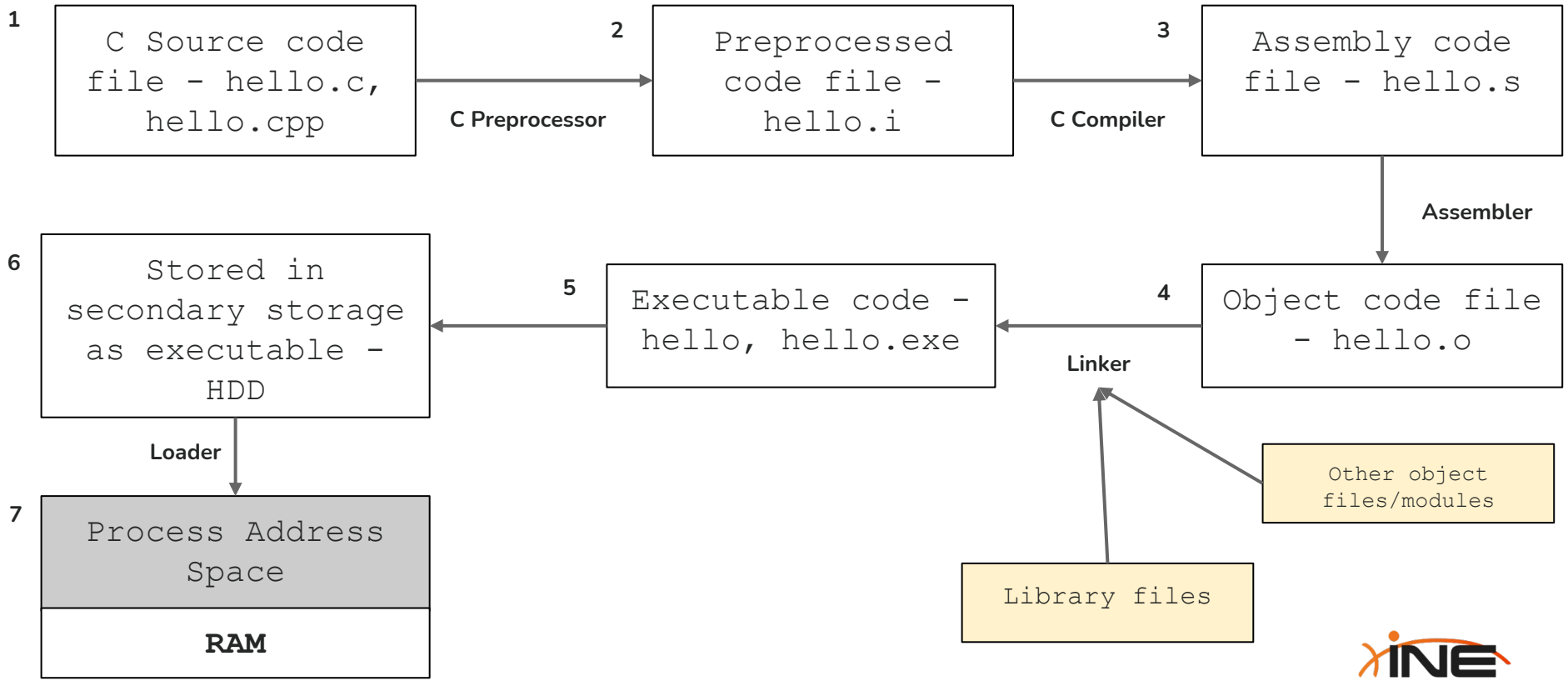
Translation

## Machine Language

```
010110100101
111010101010
101010101010
```



# Correlation With HLLs



# CPU Specific Assembly Languages

- Intel Assembly Language and ARM Assembly Language are two distinct assembly languages designed for different CPU architectures: x86 (Intel) and ARM, respectively.

**Intel Assembly Language (x86/x64)**

- Intel Assembly Language is specific to Intel's x86 and x64 CPU architectures, which are widely used in personal computers, servers, and other computing devices.

**ARM Assembly Language**

- ARM Assembly Language is specific to the ARM architecture, which is widely used in mobile devices, embedded systems, and increasingly in other computing devices.

# Intel Assembly

- Intel Architecture-specific assembly language is typically referred to simply as "Intel Assembly Language" or "x86 Assembly Language" for the 32-bit version, and "x86-64 Assembly Language" for the 64-bit version.

- These terms directly reflect the architecture of the Intel processors for which the assembly language is designed.

- Additionally, it may sometimes be referred to as IA-32 Assembly Language for the 32-bit version, where "IA" stands for "Intel Architecture." Similarly, the 64-bit version may be referred to as "IA-32e" or "Intel 64" assembly language.

# IA-32

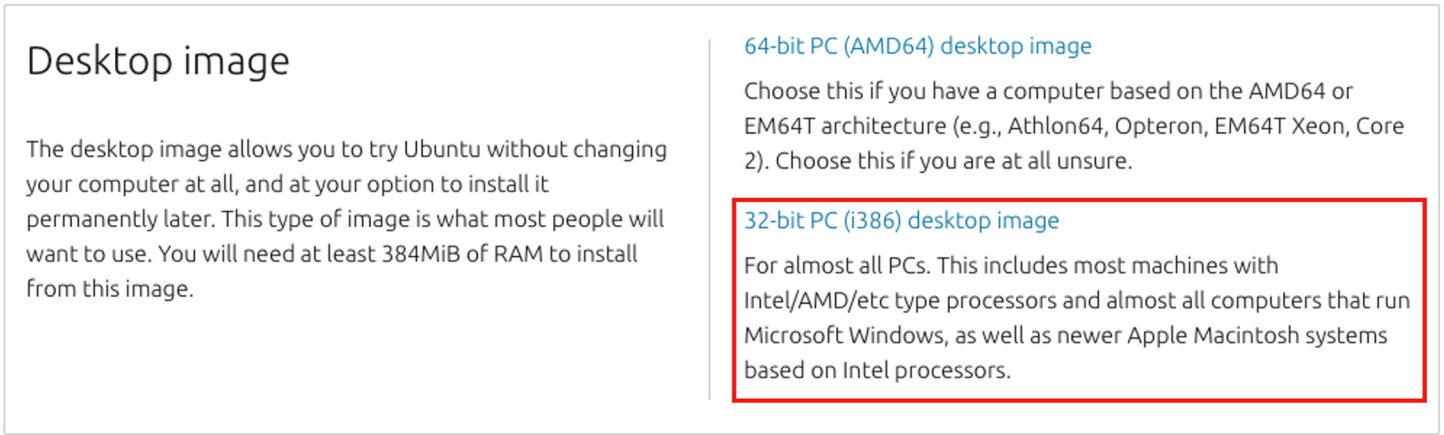
- In this course we will be using IA-32 Assembly on Linux.

- Why IA-32?
  - A large number of processors/systems are still running IA-32.
  - Learning and understanding IA-32 is the best place to start and provides a logical progression to IA-64.

# Setting Up Our Lab

# IA-32 Lab Environment

- The IA-32 exercises and demos will be performed on Ubuntu 16.04.7 LTS (32-bit) installed on VirtualBox.
  - https://releases.ubuntu.com/16.04/



# Demo: Setting Up Our Lab

# Hello World In Assembly

# Demo: Hello World In Assembly

# Data Types & Variables

# Demo: Data Types & Variables

# System Security

## Course Conclusion

# Learning Objectives:

1. Understanding Computer Architecture Fundamentals:
    - Explain the basic principles of computer architecture, including CPU architecture and process memory.
    - Understand the role of the CPU, CPU registers and memory in the operation of a computer system.
2. IA-32 CPU Architecture:
    - Describe the architecture of IA-32 processors, including registers, instruction set, and execution pipeline.
    - Understand the function and purpose of general-purpose registers, and other CPU components.
3. Memory Organization and Process Memory:
    - Explore the different memory regions, including code, data, heap, and stack, and their roles in program execution.
4. Understanding the Stack and Stack Frames:
    - Explain the stack data structure and its role in function calls and local variable storage.
    - Understand the concept of stack frames and their organization in memory during function execution.
5. Introduction to IA-32 Assembly Language:
    - Define assembly language and its role in low-level programming.
    - Differentiate between high-level languages and assembly language.
6. IA-32 Assembly Language Basics:
    - Learn how to use assembly language instructions for basic arithmetic and logical operations.
    - Understand data movement instructions and memory addressing modes in IA-32 assembly.

# Thank You!

*EXPERTS AT MAKING YOU AN EXPERT*